Implementation of Families In the Processing Graph Method Tool

by

Dick Stevens

August 29, 2002

1 Introduction

At NRL (Naval Research Laboratory) we wrote the PGM (Processing Graph Method) Specification [1] to describe the basic features of a support system for developing applications for a distributed network of processors. In the PGMT (PGM Tool) project we developed an implementation of PGM. This project was conducted with the support of the Advanced Systems Technology Office for Undersea Warfare in the Office of Naval Research. We demonstrated its ability to perform on a heterogeneous network of processors using the Unix operating system with the MPI (Message Passing Interface) communication protocol.

One fundamental concept that we use extensively in PGMT is the *family*, which we use in place of the more common *array* or *vector*. A family may be thought of as a multidimensional array with 0 or more dimensions. The number of dimensions is called the *height.*

Let F be a family with height n > 0, then

- F comprises a sequence of families with height n-1. We call each of these families with height n-1 the *children* of F. The number of children, which may be 0 or more, is called the *content* of F. If the content is 0, we say that F is *empty.*
- The lowest order elements of F (i.e., with height 0) are called *leaves* and have a common *base type.*
- If F has content = c > 0, the sequence of its families with height n-1 are indexed sequentially with indices lb, lb+1, lb+2, … , lb+c-1. The first index lb is called the *lower bound*. By default, the lower bound is 0. However in some cases, the user may choose any integer value for the lower bound. If c = 0, then the lower bound is undefined.

Now let F be a family with height 0. Then F comprises a single element of its respective base type, and its content and lower bound are undefined.

Intuitively, we may think of a family as a rooted tree. A tree with height 0 comprises a single leaf. A tree with height 1 comprises a one-dimensional array of leaves (i.e., height 0 families), each leaf having a single index to identify it. A family with height 2 comprises an array of height 1 families. In general, a family with height n comprises an array of height n-1 families. In every family at every level, the indexing has a specified lower bound. If one were to take a given family and remove the leaves, then what remains is the *family tree*.

For a complete discussion of the family concept, see the PGM Spec [1].

2 Basic Approach

A major concern in the implementation of families is how they may be used. In some cases the family structure remains fixed as long as the family exists. Specifically, all of the information needed to construct the family exists at the time of its construction. Subsequently there is no modification of the family structure until it is destroyed. We call this a *static family*. In PGM, examples of static families include the following:

- A token.
- A family of node ports or graph ports.
- A family of nodes or of included graphs.

In the case of a static family, it frequently happens that we will want to access a given leaf, given its family indices.

Another use of families is more dynamic in the sense that while the family exists, we will want to add and delete children of the family. The only example of this in PGM is the storage of tokens in a queue. PGM specifies that the tokens stored in a queue comprise a family whose children are the tokens. We call this a *dynamic family*. In this case we are less concerned about access to individual leaves and more concerned about facilitating the addition and removal of children in the family.

In PGMT we are primarily concerned about performance, and so we use different techniques for constructing and maintaining static and dynamic families.

In the case of storing tokens in a queue (i.e., a dynamic family), we use a doubly linked list to store the children in the family, each child being a token, which comprises its family structure (usually as a static family, which we discuss below). The notion of a doubly linked list is well known, and we will not discuss it further here. We devote the remainder of this section to the implementation of static families.

Using a doubly linked list to store a static family requires much memory. Moreover accessing an element by index is awkward and slow, particularly if the family height is large, requiring storage in the form of a list of lists of lists, etc. To facilitate more efficient access in a static family, we defined a special data structure called the *descriptor*.

Before we discuss the design and structure of the descriptor, we note that dynamic addition and removal of elements from this static family structure is awkward and slow. Thus we provide automatic conversion between the static and dynamic forms when it is appropriate to do so. This conversion is transparent to the user.

We provide two arrays for a given static family.

- The first array is a one-dimensional array of the leaves called the *leaf array* or *data array*. The order in which the leaves are stored in the leaf array is lexicographic, as determined by the family indexing (i.e., with the right-most index varying most rapidly).
- The second array is a one-dimensional array of integers that we call the *descriptor*, in which we capture all the information in the family tree.

From the descriptor of a given family F, we are able to calculate a wide range of different pieces of information about F. Examples are:

- Given the family indices of any leaf in F, we may calculate the index for that leaf in the leaf array.
- Given the index for a leaf in the leaf array, we may calculate its family indices.
- Given the descriptors, leaf arrays of the children, and index lower bound of a family, we may construct the leaf array and descriptor for the resulting family.
- Given the descriptor and leaf array of a family with height $n > 0$, we may construct the descriptors, leaf arrays, and index lower bound of its children.

There are several advantages to this approach; we give a few:

- The intricate and potentially convoluted structure of a family (with the number indices for each leaf as the height of the family, each index having a lower bound that is dependent on the previous sequence of indices) is reduced to a single leaf index.
- The lexicographic order of the leaves is consistent with the row-major order of most programming languages (e.g., in the case where we use a family to store a matrix).
- In MPI (Message Passing Interface), the passing of a token via a message from one processor to another is easily facilitated, as a message can be constructed that comprises three parts: A header that provides information about the message, the descriptor (an integer array), and the leaf array.

3 Structure of Descriptor

As noted above and in the PGM Spec [1], a family with height 0 is equivalent to a single leaf. Thus, to unify the management of structures in PGMT, we consider that almost every structure belongs to a family. If the family has height 0, in which case there is a single leaf, we say that it is a *trivial family*. Conversely, a family with height > 0 is a *non-trivial family*.

We define a *descendant* of a family recursively: A descendant of a family F is a family that is either a child of F or a descendant of a child of F. Intuitively, a descendant of F is a child of F, a child of a child of F, … , or a leaf of F.

In a non-trivial family, if all the children have identical family trees, we say that the family is *regular*. Otherwise we say that the family is *irregular*. A family is *completely regular* if it is regular and all its descendants are regular. Note that a family can be regular with an irregular child, thus not being completely regular.

In simple terms, a descriptor has a recursive structure. At the beginning of the descriptor are two segments called the *preamble* and the *header*. These two segments contain some preliminary information that indicates the structure of what follows:

- If the family is trivial, the rest of the descriptor is empty.
- If the family is non-trivial, then the rest of the descriptor depends on whether the family is regular:
  - If the family is regular, then what follows is the common descriptor of the children.
  - If it is irregular, then what follows is the concatenation of the descriptors of the children.

We anticipate that most non-trivial families will be completely regular. Compared to the descriptor of an irregular family of comparable size, the descriptor of a completely regular family is greatly compressed, thus saving memory. An added benefit of this compression is that some of the functions for obtaining information about the descriptor structure are much faster.

We now discuss the detailed structure of the descriptor.

The first two integers in the header make up the *preamble*. These two integers give the family height and the number of leaves. For a trivial family, the preamble is the entire descriptor, specifying family height 0 and 1 leaf.

As stated above, a non-trivial irregular descriptor contains a concatenation of the descriptors of its children, and a non-trivial regular descriptor contains a single copy of the common descriptor of its children. In both cases, we omit the preambles from the children's descriptors, because the height and number of leaves for each child can be calculated.

Following the preamble is the header, which has between 2 and 4 integers. The first two integers of the header are present in every non-trivial descriptor, and the third and fourth integers are optional. The following integers are in the header:

1) Status word whose bits indicate information about the form of the descriptor:
   a. Bit 0: set if the family is regular (i.e., all children have identical family trees).
   b. Bit 1: set if the lower bound is 0.
   c. Bit 2: set if the stride is specified in the header, i.e., if all the following are true:
      i. family is regular,
      ii. leaf count > 0, and
      iii. height > 1.

    d. Bit 3: set if the stride is not specified and is 0.
2) The number of children.
3) The *stride*. The stride is the common number of leaves in each child of a regular family. It is present only for non-empty regular families with height > 1.
   Notes:
    a. A family with height 1 is necessarily regular, and each child has height 0, implying that its stride is 1. Bit 2 in the status word indicates that the stride is present in the descriptor.
       NOTE:
    b. If the shape is regular and there are no leaves, then the stride is zero but is not specified: hence the reason for bit 3 of the status word.
4) The lower bound of indices for the children.
   Notes:
    a. The lower bound is present if and only if the lower bound is not 0 – i.e., if bit 1 of the status word is not set.
    b. If the stride is not specified in word 3, then the lower bound (if specified) is in word 3.

In an irregular descriptor, recall that not all children have the same descriptor. In this case we concatenate all of the descriptors of the children (the preamble of each child being omitted as redundant). After the header and before the concatenated descriptors of the children, there is a sequence of pairs of integers, one pair for each child. The first of each pair is the relative index for the start of the child's data in the data array, and the second of each pair is the relative index for the start of the header of the child's descriptor. The concatenated descriptors follow this sequence of pairs. NOTE: The descriptor of each child is copied verbatim (without the preamble) from the descriptor of the respective child. Thus, the indices in each pair are relative to the respective beginnings of the data array and descriptor being constructed.

With the information thus contained in the descriptor, we define the following functions for access to a family:

- to return the lower bound of indices of the children
- to return the number of children
- to return the number of leaves
- to return the height
- to return a list of the descriptors of the children
  (this function is used, for example in the unpack transition)
- given family indices that identify a leaf, to return the single index in the data array
- given the single index of a leaf in the data array, to return the family indices

In addition, there are several functions to be used for constructing a descriptor:

- to construct a descriptor for the trivial family

- given family indices that identify a descendant family, to construct the descriptor for that descendant family
- given a family height, to construct a descriptor for an empty family with the given height
- given a lower bound for the family index and number of children, to construct a descriptor for a height 1 family with the given lower bound and number of children
- given an array of integers specifying the size in each dimension, to construct the descriptor of a regular family with the respective size in each dimension (index lower bound = 0 in all dimensions)
- given a list of descriptors, construct the descriptor of the family whose child descriptors are in the given list (this is used, for example in the pack transition)

Having described how families are implemented in PGMT, we say a few words about the motivation behind the overall design of PGMT. The human user may use families of nodes and families of included graphs, each possibly having one or more families of ports. All of this represents a complex structure that is meaningful to the user.

The purpose of families and included graphs is to provide leverage for the user to specify a large graph involving repeated constructs with similar structure. A family of nodes or of included graphs allows one to specify a generic description of part of a PGM graph to be replicated any number of times during graph construction.

4 Example

Having given the basic ideas of a descriptor, we proceed to show how to construct the descriptor for the family example shown in the PGM Spec [1].

We start with the descriptors of the first height 1 family, denoted by F[0][9]. The data array is {39,40,–41}, and the descriptor is {1,3,1,3,–1}, which we explain in Table 1. The next height 1 family F[0][10] has an identical family tree, also shown in Table 1. Its data array is {-49, 50, 51}.

For the height 2 family F[0], which is regular, the data array is the concatenation of the data arrays for F[0][9] and f[0]10]: {39, 40, –41, –49, 50, 51}. Table 2 gives the descriptor.

index	value	segment	word	remarks
0	1	Preamble	height	
1	3		leaf count	
2	1	Header	Status word	regular, lower bound ≠ 0,

				stride not given, stride ≠ 0
3	3		child count	
4	-1		lower bound	

Table 1

index	value	segment	word	remarks
0	1	Preamble	height	
1	6		leaf count	
2	5	Header	status word	regular, lower bound ≠ 0, stride given, stride ≠ 0
3	2		child count	
4	3		stride	
5	9		lower bound	
6	1	Descriptor	status word	common descriptor of children without preamble
7	3		child count	
8	-1		lower bound	

Table 2

The family F[1] is empty with height 2. There are no leaves, and so the data array is empty. Table 3 gives its descriptor.

index	value	segment	word	remarks
0	1	Preamble	height	
1	0		leaf count	
2	11	Header	status word	regular, lower bound = 0, stride not given, stride = 0
3	0		child count	

Table 3

Starting to do the third height 2 family F[2], we do its height 1 children in order. First, F[2][0] has the data array {0, 1, 2, 3} and descriptor in Table 4:

index	value	segment	word	remarks
0	1	Preamble	height	
1	4		leaf count	
2	3	Header	Status word	regular, lower bound = 0, stride not given, stride ≠ 0
3	4		child count	

Table 4

F[2][1] has the data array {9, 10, 11} and has descriptor in Table 5:

index	value	segment	word	remarks
0	1	Preamble	height	
1	3		leaf count	
2	1	Header	Status word	regular, lower bound ≠ 0, stride not given, stride ≠ 0
3	3		child count	
4	-1		lower bound	

Table 5

F[2][2] is empty and has descriptor in Table 6:

index	value	segment	word	remarks
0	1	Preamble	height	
1	0		leaf count	
2	3	Header	Status word	regular, lower bound = 0, stride not given, stride ≠ 0
3	0		child count	

Table 6

F[2][3] has the data array {28, 29} and has descriptor in Table 7:

index	value	segment	word	remarks
0	1	Preamble	height	
1	3		leaf count	
2	1	Header	Status word	regular, lower bound ≠ 0, stride not given, stride ≠ 0
3	3		child count	
4	-1		lower bound	

Table 7

Having given the data arrays and descriptors for all of the children of F[2], we now give the data array and descriptor for F[2] itself. The data array is the concatenation of the data arrays of the children: {0, 1, 2, 3, 9, 10, 11, 28, 29}. The descriptor is in Table 8:

index	value	segment	word	remarks
0	2	Preamble	height	
1	9		leaf count	
2	2	Header	Status word	not regular, lower bound = 0, stride not given, stride not defined
3	4		child count	
4	0	offset pairs	data offset	first child
5	10		descriptor offset	
6	4		data offset	second child
7	12		descriptor offset	
8	7		data offset	third child
9	15		descriptor offset	
10	7		data offset	fourth child
11	17		descriptor offset	
12	3	descriptors	descriptor	first child (table 4)
13	4			
14	1		descriptor	second child (table 5)
15	3			

16	-1			
17	3		descriptor	third child (table 6)
18	0			
19	3		descriptor	fourth child (table 7)
20	2			

Table 8

index	value	segment	word	remarks
0	3	Preamble	height	
1	15		leaf count	
2	2	Header	Status word	not regular, lower bound = 0, stride not given, stride not defined
3	3		child count	
4	0	offset pairs	data offset	first child
5	8		descriptor offset	
6	6		data offset	second child
7	15		descriptor offset	
8	6		data offset	third child
9	17		descriptor offset	
10	5	descriptors	descriptor	first child (table 2)
11	2			
12	3			
13	9			
14	1			
15	3			
16	-1			
17	11		descriptor	second child (table 3)
18	0			
19	2		descriptor	third child (table 8)
20	4			
21	0			
22	10			
23	4			
24	12			
25	7			
26	15			

27	7			
28	17			
29	3			
30	4			
31	1			
32	3			
33	-1			
34	3			
35	0			
36	3			
37	2			

Table 9

The data array for F is the concatenation of the data arrays for its children F[0], F[1], and F[2]: {39, 40, –41, -49, 50, 51, 0, 1, 2, 3, 9, 10, 11, 28, 29}. Table 9 shows its descriptor.

5 Access Functions

We define a number of functions to support their construction and maintenance as well as to provide access for the user. In addition to conserving the integrity of the descriptors, these functions also allow the user access to the family structure without having to be concerned about the details of the descriptor.

In the following descriptions of access functions, the language is C++, and the class is GCL_Descriptor. These functions are public class methods of this class; thus the user may call any of these functions. We give the prototype (i.e., the function name and its argument sequence) and a brief description for each function. Note, the terms depth and height have the same meaning and will be used interchangeably.

For complete details, refer to the source code[2]. The source is well documented, albeit with some terminology that differs from this paper.

First, we give several constructors and the destructor.

```
/* default constructor:  constructs a descriptor for a family with height 0; i.e., descriptor of a scalar */
GCL_Descriptor ();

/* copy constructor:  makes a new descriptor that is a copy of the given descriptor */
GCL_Descriptor (const GCL_Descriptor & descriptor);

/* constructor of a descriptor of a descendant of a given descriptor:
```

```
        uses the indices to identify the descendant and constructs the descriptor of the descendant */
    GCL_Descriptor (const GCL_Descriptor & descriptor,
                                    const_vector_int & indices);

    /* constructor: descriptor for empty family with given height */
    GCL_Descriptor (unsigned int inDepth);

    /* constructor with given lower bound and child count, height 1 */
      GCL_Descriptor (int lowerBound, unsigned int childCount);

    /* constructor of completely regular descriptor */
      GCL_Descriptor (const vector<unsigned int> & dimensions,
                int lowerBound = 0);

    /* constructor with given deque of children
          (A deque (double ended queue) is defined in the Standard Template Library) */
      GCL_Descriptor (const deque<GCL_Descriptor *> & indeque,
                unsigned int inDepth, int lowerBound = 0);

    /* destructor */
      ~GCL_Descriptor ();
```

Here are the public access functions:

```
    /* overloaded assignment operator: returns a copy of the given descriptor */
      void operator = (const GCL_Descriptor & descriptor);

    /* to determine whether another descriptor equals this one:
          returns true if the descriptors are equal (i.e., describe the same family tree) and false if not */
      bool equal (const GCL_Descriptor & descriptor) const;

    /* return the lower bound */
      int getLowerBound() const;

    /* return the number of children */
      unsigned int getChildCount() const;
```

```
/* return the number of leaves */
  unsigned int getLeafCount() const;

/* return the depth (i.e., the height) */
  unsigned int getDepth() const

/* return a deque of descriptors of the children: useful for unpacking a family */
  deque<const GCL_Descriptor *> * makeChildren() const;

/* return the data offset corresponding to given index vector:
      useful for finding where the data begins for a leaf or descendant whose indices are known */
  unsigned int getOffset(const_vector_int & indices) const;

/* return the index vector corresponding to given leaf offset:
      useful for determining the user's indices for a given leaf when the data offset is known */
  vector<int> * getIndices(unsigned int offset) const;
```

Finally, here are some non-class functions for access to descriptors:

```
/* overloaded predicate operator: return true iff left and right are equal descriptors  */
  bool operator == ( const GCL_Descriptor & left,
      const GCL_Descriptor & right);

/* overloaded predicate operator: return true iff left and right are not equal descriptors  */
  bool operator != ( const GCL_Descriptor & left,
     const GCL_Descriptor & right);



/* function to assemble a descriptor from a parent descriptor and a
   deque of child descriptors */
  const GCL_Descriptor * assemble
     (const GCL_Descriptor & parent,
      const deque <const GCL_Descriptor *> & children,
      unsigned int childDepth);

/* function to disassemble a descriptor into a parent descriptor and
   a deque of child descriptors */
  const GCL_Descriptor * disassemble
     (const GCL_Descriptor & descriptor,
```

```
deque <const GCL_Descriptor *> & children,
unsigned int childDepth);
```

6 References

1. *Processing Graph Method 2.1 Semantics*, by David J. Kaplan and Richard S. Stevens, July 29, 2002.
2. Graph Class Library (GCL) Source Code files GCL_Descriptor.h and GCL_Descriptor.cpp.